Drag-and-Drop Refactoring: Intuitive Program Transformation

Yun Young Lee, Nicholas Chen, Ralph Johnson
University of Illinois at Urbana-Champaign
{lee467, nchen, rjohnson}@illinois.edu

ABSTRACT

Refactoring is a disciplined technique for restructuring code to improve its readability and maintainability. Almost all popular integrated development environments (IDEs), such as Eclipse, Visual Studio, and Xcode, have built-in support for semi-automated refactorings. Proponents tout that semi-automated refactorings reduce the burden of refactoring by hand. However, recent research suggests that these semi-automated refactorings are greatly underused. We argue that the current semi-automated refactoring tools are complex to use, which could be one of the causes of their underuse. In this paper, we present a novel approach that reduces this complexity by streamlining the invocation and configuration process through drag-and-drop of program elements. We implemented this approach in our tool, *Drag-and-Drop Refactoring* (DNDRefactoring). Currently, DNDRefactoring supports 12 of 23 refactorings in the Eclipse IDE. Empirical evaluation through surveys (69 results) and controlled user studies (11 participants) demonstrates that DNDRefactoring is intuitive and also reduces the programming effort compared to traditional methods such as menus and keyboard shortcuts.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Coding Tools and Techniques; H.5.2 [**User Interfaces**]: User-centered Design

General Terms

Design, Human Factors

Keywords

Software, refactoring, restructuring, drag-and-drop

1. INTRODUCTION

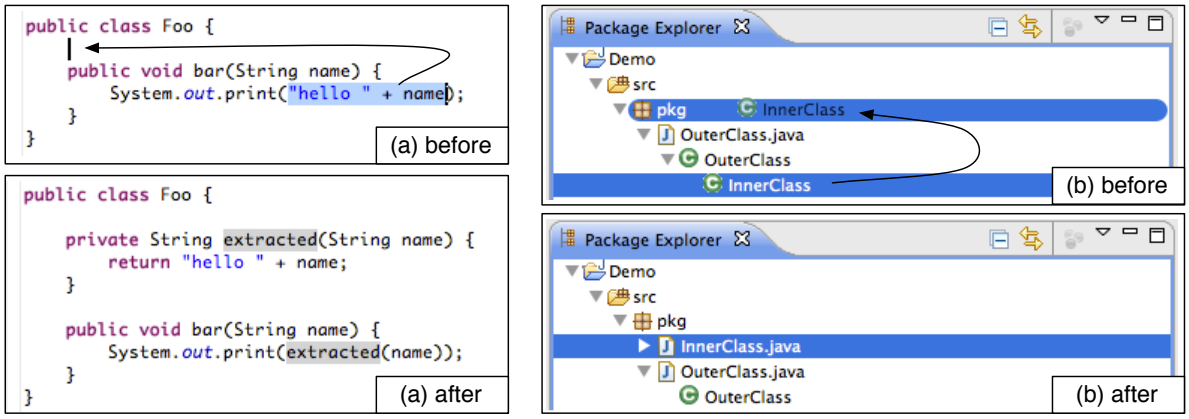
Refactoring is a *disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior* [13]. The concept of refactoring has been much studied and improved since it was first introduced by Opdyke and Johnson in 1990 [23], and is now a well-accepted programming practice. Almost all popular IDEs, such as Eclipse, IntelliJ, Visual Studio, and Xcode, include semi-automated refactoring tools. While no IDE supports all 72 refactorings that Fowler cataloged in his book [14], the number of refactorings that IDEs support has only been increasing. For example, Eclipse 2 (as of 2004) supported 14 refactorings [15] but the most recent version of Eclipse (Indigo) contains 23 refactorings. Just as the refactoring tools in IDEs have been improving, there also has been much research analyzing their usage. Murphy-Hill et at. [20] analyzed Eclipse refactoring tool usage and concluded that almost 90% of refactorings are performed manually without the help of the tool, and when programmers *do* use the tool, 90% of them do not modify its default configurations. Vakilian et al. reported that programmers, on average, are unaware of more than nine existing refactorings in Eclipse, and find some refactorings to have names that are difficult to associate with actual refactorings [26].

Several efforts have been attempted to help promote the use of refactoring tools. Alexis et al. [22] and Parnin et al. [24] introduced tools that provide visual cues in the IDE to help programmers identify opportunities for performing refactorings. Murphy-Hill et al. introduced tools that assist programmers when they invoke refactorings in Eclipse, with visual selection assists [18] and gesture-to-refactoring mappings [21]. Although such additional tools *could* encourage programmers to use refactoring tools, they inadvertently add extra layers of interface. These extra layers unnecessarily increase the complexity and effort required from programmers to use refactoring tools – the same tools that are already underused in the first place. *What if these layers of interface are not added, but stripped away?*

Answering this question led to our novel approach of using drag-and-drop. Almost all popular IDEs support drag-and-drop, but mostly for textual cut-and-paste. To the best of our knowledge, our work is the first to exploit it as an effective invocation mechanism for refactoring tools. We also present *Drag-and-drop Refactoring* (DNDRefactoring), our tool implementing the drag-and-drop paradigm. DNDRefactoring aims to promote the use of refactoring tools, by offering a simple and intuitive mechanism for invoking many of the popular refactorings inside IDEs without the overhead of additional layers of interfaces. Specifically, DNDRefactoring allows programmers to invoke existing refactorings in IDEs via drag-and-drop of the program's abstract syntax

FSE '12 Research Triangle Park, North Carolina USA

Figure 1: Drag-and-drop gestures in (a) Java editor for "Extract Method" refactoring, and (b) Package Explorer for "Extract Type to New File" refactoring.



tree (AST) elements (see Figure 1a).

This paper makes the following contributions:

1) Technique: Our minimalistic invocation mechanism relies on drag-and-drop of AST elements. The invocation mechanism depends on the *drag source* and the *drop target* of AST elements. As such, it is generalizable to different refactorings and different programming languages. We introduce the *first* canonical set of drag-and-drop gestures for 12 refactorings for Java. Tables 1 and 2 detail the drag source and drop target for the supported refactorings.

2) Tool: We implemented our technique in an open source tool, DNDRefactoring, for the Eclipse IDE. DNDRefactoring is supported i) within a Java editor, or ii) within and between Package Explorer and Outline views[1].

3) Evaluation: We evaluate the intuitiveness and utility of our technique and tool in two ways. First, we surveyed software engineering students asking them to map drag-and-drop to refactorings and vice versa, to measure the intuitiveness of our canonical set of drag-and-drop gestures. Second, we conducted a controlled experiment and user study to evaluate how DNDRefactoring can assist programmers in terms of reducing their programming effort. The results show that DNDRefactoring is not only intuitive but also reduces programming effort by up to 37.8% compared to traditional refactoring tools.

2. DESIGN RATIONALE

Several researchers have introduced tools in the past that either directly or indirectly promote the semi-automated refactoring tool usage in IDEs. These tools, however, also introduce additional interfaces that programmers must understand and interact with. We believe that the added layer of interface can easily add complexity to already underused refactoring tools. Therefore, the basic premise of our design was to minimize the addition of interfaces and thus reduce the programming effort required from programmers. Four main factors influence our decision to use drag-and-drop:

1) Consistency: Drag-and-drop *standardizes* the invocation mechanism between different refactorings. Programmers no longer have to search through lengthy lists of menu items or memorize ornate keyboard shortcuts for individual refactorings. With drag-and-drop, there is only one consistent invocation mechanism: selecting a source and a target AST elements. The source and target automatically determine which refactoring to invoke.

2) Universality: Drag-and-drop refactoring is applicable for many programming languages. All that is required is an IDE that is aware of the underlying AST and provides some mechanism for interacting with the individual AST elements. Most modern IDEs already provide some form of drag-and-drop support. For instance, drag-and-drop gestures in Eclipse's editors perform textual cut-and-paste; drag-and-drop in the Package Explorer perform simple move refactorings. It is natural and logical to extend the underlying infrastructure to support more refactorings.

3) Programmers in control: Drag-and-drop allows programmers to perceive written code as a malleable entity that they can transform directly and intuitively, instead of a flat textual representation of the program. We believe that drag-and-dropping the AST elements directly to perform refactorings, as opposed to static menus and buttons, gives programmers the impression that they are in control of the changes instead of IDEs performing refactorings behind the scene. Refactoring by drag-and-drop of AST elements also aligns well with the object-oriented programming paradigm as programmers are able to visualize a class as an entity composed of smaller components such as methods and fields.

4) Streamlined changes: Drag-and-drop streamlines the invocation of refactoring tools by eliminating all forms of unnecessary configurations. In particular, drag-and-drop relies on sensible default configurations. This eliminates the modal windows for user inputs and configurations during the refactoring process. As stated earlier, if 90% of refactoring tool users do not modify the default configurations [20], the pop-ups are nothing more than interruptions to programmers. Eclipse already provides a workaround for this issue with *Quick Assist* [2], which performs local refactorings with default values and then allows programmers to make in-line changes. We believe that such *lazy configuration* delays and reduces decision-making for programmers thus making the tool simpler to use. DNDRefactoring leverages the Quick Assist paradigm whenever possible.

[1]The Package Explorer and Outline views show a Java element hierarchy tree of the Java projects and source files.

Table 1: Refactorings with Drag-and-Drop: within a Java editor.

Drag Source	Drop Target	Refactoring
Local variable	Declaring type	Promote local variable to field (ILE[1])
Expression inside method	Same method	Extract temp variable (ILE)
	Between argument brackets of current method signature	Introduce parameter
	Declaring type	Extract method (ILE)
Statements in method	Declaring type	Extract method (ILE)
Non-static method	Field variable in declaring type	Move instance method to field type
	Argument type in current method signature	Move instance method to argument type
Static method of field	Another type in current editor	Move member to target type
	Field variable in declaring type	Move member to field type
	Local variable type in declaring type	Move member to local variable type
Anonymous class	Declaring type	Convert anonymous to nested type

[1] ILE = In-Line Edit allowed after refactoring is completed.

3. DND REFACTORING FEATURES

Out implementation of DNDRefactoring in Eclipse does not implement any new refactorings or alter the functionality of existing ones, but only provides programmers new methods of invocation. More specifically, it allows programmers to invoke existing refactorings by drag-and-dropping Java AST elements 1) within the Java editor or, 2) within and between the Package Explorer and Outline View. The drag source is the highlighted selection, either a text selection within a Java editor or a tree node in the Package Explorer or Outline View. The drop target is identified by the position of the cursor when the drag source is dropped. For example, within a Java editor, a cursor located in a whitespace anywhere inside a class, but outside any method and not in any field declaration, will identify the target as the class (Figure 1a). A refactoring is invoked based on the Java AST element types of the drag source and drop target. If no suitable refactoring is found, the drag-and-drop gesture defaults to the textual cut-and-paste function.

Tables 1 and 2 list all the drag-and-drop refactorings that we have implemented for the Eclipse IDE. To the best of our knowledge, the mappings in the tables are *new* and serve as the *first* canonical set of drag-and-drop gestures for refactorings. Other mappings for the stated refactorings are certainly possible, but the current mappings were determined based on an iterative design process made by all the authors. Also, the stated refactorings are not the only ones that can be supported by DNDRefactoring. For example, one can support "Inline Local Variable" refactoring by selecting a local variable and dropping in a location where it is used. However, we felt such mapping between the refactoring and drag-and-drop gesture was less intuitive, and limited our canonical set to move and extract-based refactorings. Section 5 presents empirical evaluations to support the intuitiveness of those gestures.

3.1 Extended Features

In addition to providing a new method of invocation, DNDRefactoring also supports two new and useful features that can only be accomplished through drag-and-drop gestures.

1) Collated refactorings: A single drag-and-drop gesture can effectively collate several refactorings together. Consider dragging a nested class and dropping it in the *current* package. This gesture can be translated into the "Move type to New File" refactoring in Eclipse. What happens if the nested class was dropped in a *different* package? Naturally, the gesture of dropping the nested class onto a different package can be interpreted as "Move type to New file" + "Move type to target package". This collated refactoring is intuitive and effortless using DNDRefactoring. And, yet, such a simple collated refactoring is impossible to invoke using the existing invocation and configuration mechanisms in Eclipse. Programmers using the traditional invocation mechanisms are forced to perform two separate refactorings in succession. Support for collated refactorings in DNDRefactoring intuitively combine refactorings together in a single drag-and-drop gesture. Collated refactorings are annotated with "+" in Table 2.

2) Precise control: Another advantage of drag-and-drop is the ability to precisely choose where a drag source is dropped. Consider the refactoring "Extract Method". Currently the "Extract Method" refactoring in Eclipse always creates a new method *below* the method from which the expression or statements were extracted. However, with DNDRefactoring, programmers' natural expectation would be to see the extracted method appear exactly where the expression was dropped (see Figure 1a). DNDRefactoring supports this requirement and allows programmers to decide where to move or extract Java AST elements. We believe that this feature strengthens our design goal of giving programmers more control over their code.

3.2 Supporting Floss Refactoring

Murphy-Hill and Black introduced the term *floss refactoring* to describe refactorings that occur frequently in small steps, intermingled with other kinds of program changes [19]. They also proposed five principles to characterize a tool that supports floss refactoring. They suggest that such tools should let the programmer:

1. choose the desired refactoring quickly,

Table 2: Refactorings with Drag-and-Drop: within and between Package Explorer and Outline View.

Drag Source	Drop Target	Refactoring
Non-static Method	Type of field variable in declaring type	Move instance method to target field type
	Type	Pull-up, Push-down or Move method to target type
Nested Type	Package	Move nested type to new file + Move type to target package
Anonymous Type	Type	Convert anonymous to nested type
	Package	Convert anonymous to nested type + Move nested type to new file + Move type to target package
Field	Type	Pull-up, Push-down or Move field to target type
Static Members	Another type declared in current editor	Move members to target type
	Type of field variable in declaring type	Move members to target field type
	Type of local variable in declaring type	Move members to local variable type
Non-static fields	Package	Extract data class + Move type to target pacakge
Non-static methods	Package	Extract interface
Static & non-static methods	Package	Extract super class

2. switch seamlessly between program editing and refactoring,
3. view and navigate the program code while using the tool,
4. avoid providing explicit configuration information, and
5. access all the other tools normally available in the development environment while using the refactoring tool.

Current refactoring tool in Eclipse violates all five principles [19]. The refactoring tools by Murphy-Hill et al. help programmers' code selection process 1) with syntactic highlights, 2) by visualizing nested statements as a series of nested boxes, and 3) with control- and data-flow annotations [18]. While the tools were proven to help reduce time and errors during refactoring, they violate Principles 1 and 4 because the tools do not assist programmers with refactoring selection or configuration. The same limitation applies to tools that alert programmers of code smells and thus opportunities for refactorings [22] [24]. Murphy-Hill et al. introduced other tools that help with refactoring selection, by mapping gestures to refactorings [21]. The tool displays a radial menu with four quadrants, and maps directional gestures (up, down, left or right quadrants) to refactorings. The tool adheres to Principles 1 and 4 because the radial menu displays a more concise set of applicable refactorings and performs the selected refactoring without requiring explicit configuration from programmers. However, the radial menu is a modal pop-up menu that can cover up part of the Java editor and thus violates Principles 2 and 3.

In contrast, we claim that DNDRefactoring satisfies all five principles. DNDRefactoring eliminates the need for programmers to browse through a long list of refactoring menu items and decode refactoring names that aren't always obvious, therefore Principle 1 is satisfied. In addition, because programmers choose source and target AST elements in the editors and views that they are currently working on, Principles 2 and 3 are satisfied. DNDRefactoring does not show modal windows during refactoring, so it also adheres to Principles 5. Lastly, DNDRefactoring also adheres to Principle 4 because it does not interrupt refactoring processes with pop-up prompts, but uses default values to complete the refactoring and then invites programmers to make in-line changes.

Code smell detection tools are perhaps more suitable for root-canal refactoring, which is an infrequent, larger scale refactoring intended for removing code smells without adding new functionality to the program [19]. While DNDRefactoring is useful for floss refactoring, its utility is not limited only to floss refactoring.

4. EVALUATION METHODOLOGY

To evaluate the utility of DNDRefactoring, we ask the following research questions:
RQ1 How intuitive is our proposed drag-and-drop gestures for DNDRefactoring? (Section 4.1)
RQ2 Does DNDRefactoring reduce programming effort required from users, compared to invoking refactorings through menus and keyboard shortcuts? (Sections 4.2 and 4.3)
RQ3 Would programmers use DNDRefactoring for their daily programming tasks? (Section 4.3)

Our tool and its tutorial video, all survey and study materials, and results are available at `https://wiki.engr.illinois.edu/display/cs599yyl/DND+Refactoring`.

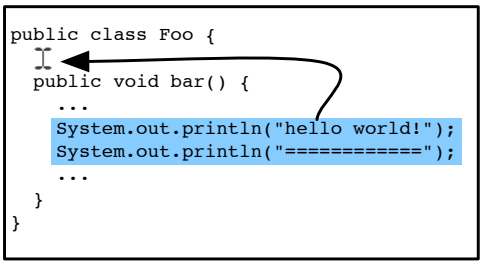
4.1 Survey

The survey contains five questions asking participants to guess a refactoring given a drag-and-drop gesture, and five questions asking the reverse mapping. The questions were given in English sentences accompanied by figures (Figure 2). We also collected data regarding survey participants' experience with Java, Eclipse and the refactoring tool in Eclipse. Participants were given a 5-minute summary of the study and were asked to complete the survey in 10 minutes.

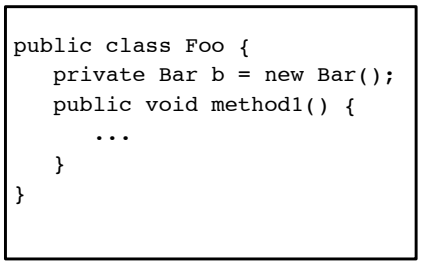
We used our canonical set of mappings (Tables 1 and 2) as an oracle, and compared the participants answers to our own mappings. A high percentage of matches would thus indicate that the mappings are intuitive for programmers. The survey results also served as a measure of feasibility before controlled user study was conducted.

The survey was conducted in a graduate-level software en-

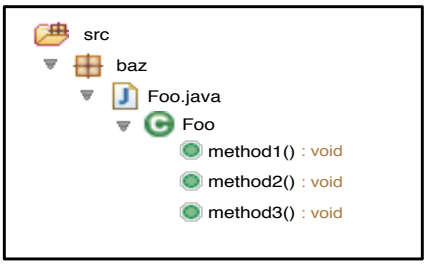
Figure 2: Examples of Survey Questions



a) Mapping from drag-and-drop to refactoring: What refactoring would you expect to invoke with the following drag-and-drop gesture?



b) Mapping from refactoring to drag-and-drop: How would you move Foo.method1() to Bar class?



c) Mapping from refactoring to drag-and-drop: How would you extract an interface to baz package with method1 and method2?

gineering class. The course teaches software development, management, and maintenance, and at least 95% of the students have taken a prerequisite course in the previous semesters that familiarizes them with Java and Eclipse. The survey was completely voluntary and anonymous. The survey results are detailed in Section 5.1.

4.2 Lab Experiment

Measuring programming effort was a challenge. How do we define and measure programming effort? Programming is inherently mental labor, but not only is the amount of mental effort needed extremely difficult to measure, it can vary greatly between programmers depending on their experience and competence. Programming, however, is also physical labor because programmers write code by typing and interacting with IDEs with a keyboard and a mouse. If programming is purely mental labor, programmers would be able to write code just by thinking. Therefore we concentrated on measuring physical effort required during programming and recorded the keyboard and mouse usage of programmers: keyboard stroke counts, mouse button counts, and the total distance that mouse movements cover. We collected these data using WhatPulse software [8]. The overall time it takes for programmers to complete the refactoring tasks were not measured, however, as refactorings are not time-sensitive operations. We hypothesized that DNDRefactoring will require less programming efforts compared to existing refactoring invocation methods.

We conducted a laboratory experiment in order to test our hypothesis and provide lower bounds for metrics we are measuring. Using Eclipse, the first author completed the same refactoring tasks used in the user study in a strictly mechanical manner. Firstly, she repeated the tasks three times, once each for different refactoring invocation methods available in Eclipse: refactoring menu, keyboard shortcuts, and DNDRefactoring. We did not include Quick Assist in our lab experiment because "Extract Local Variable" and "Convert Anonymous Class to Nested" refactorings were the only relevant ones that Quick Assist supports. Secondly, any mistakes such as typing errors, redundant mouse button clicks, and unnecessary mouse movement, were minimized as much as possible. Such mistakes were either subtracted at the end or the experiment was discarded and repeated. Also, each refactoring was performed in a way that minimizes programming effort. Anything that can be configured in the modal window (e.g. access modifier of an extracted method) was configured in the window, at the earliest time possible. For keyboard shortcut invocation method, the shortcut keys were explicitly set such that there were no duplications in the Eclipse workspace. Thirdly, the first author performed the refactoring tasks as fast as she could in order to further minimize any noise, but in a way she would normally do such tasks. She uses a mouse in her day-to-day programming, so she used a mouse during the lab experiment unless there were shortcut keys she knew of and was comfortable with. Therefore the combinations of mouse and keyboard uses may not have achieved the most minimal effort possible, but we are certain that it is at least as minimal as an average programmer could achieve. Fourthly, the user study was broken down into individual refactoring tasks and each refactoring task was repeated three times per invocation method in order to normalize the data. The lab experiment results are described in Section 5.2.

4.3 Controlled User Study

The refactoring tasks given to the participants is based on the Refactoring Lab Session exercise developed at LORE [12]. The exercise involves the refactoring of a Local Area Network simulation program. We made minor modifications to the refactoring tasks in order to remove duplicated refactorings and include a wider variety of refactorings. All 11 participants were asked to carry out the refactoring tasks twice, once with an unchanged version of Eclipse and once with a version of Eclipse including DNDRefactoring feature patch (for brevity, referred to Eclipse and DNDRefactoring, respectively, from here on), the order of which was randomized. 5 participants used DNDRefactoring first. All participants were given a group tutorial on DNDRefactoring at lease three days prior to individual user studies. The tutorial on Eclipse's refactoring tool was left optional, since the participants are already fairly experienced and familiar with Eclipse, but the two novice users were explicitly given an online resource to recap the basics of Eclipse's refactoring tool [3]. All participants were also given as much time as they needed to familiarize themselves with the given code and instructions before starting the user study. After the user study, each participant was asked to complete a post-study qualitative survey to evaluate their experience with DNDRefactoring.

We observed each of the user study sessions without being involved, but gave hints if participants made mistakes at least three times repeatedly, for example, selecting an in-

correct refactoring from the menu or selecting wrong drag source or drop target. We also recorded these failed attempts as they are indicative of how difficult each invocation method is.

We had 11 user study participants, all computer science graduate students majoring in various subdesciplines, including software engineering and software testing. The participants, on average, had 6.5 years of Java experience and 4 years of Eclipse experience. 2, 7, and 2 participants regarded themselves as novice, intermediate and expert users of the Eclipse refactoring tool, respectively. Participation was strictly voluntary with no rewards offered, and invitations to the study was sent through individual emails and departmental mailing lists. Every participant used their own computer or laptop for the user study, as using a different and possibly unfamiliar machine designated for the user study can affect their performance in many ways. We used a screencasting software or a video camera to record the entire duration of each participant's session for better accuracy of records. The user study results are described in Section 5.3.

5. RESULTS AND OBSERVATIONS

5.1 Survey Results and Observations

We collected 82 survey responses in total. We then discarded 13 surveys that had 5 or more unanswered questions which we considered as incomplete surveys. We base our analysis on the remaining 69 survey responses (Figure 3). Of those 69 participants, 57 participants (83%) indicated that they have more than 2 years of Java experience, and 55 participants (80%) have more than 2 years of experience with Eclipse. Also, 14 (20.2%), 50 (72.5%), and 5 (7.3%) participants regarded themselves as novice, intermediate, and expert users of the refactoring engine in Eclipse, respectively.

The first and second authors each graded half of the responses, and compared their grading schemes in order to minimize bias. Most of participants' answers were straight forward to determine correct or incorrect, and the two authors discussed and agreed on unclear answers. For example, at least 9 participants answered "Extract Class" or "Push Up Class" for a drag-and-drop gesture depicting the "Move Type to New File" refactoring in Eclipse (Figure 1b). Even though such answers seem to convey the action of the actual refactoring, they were graded to be incorrect as the names conflicted with other existing refactorings in Eclipse. Participants' answers to each question were marked either correct, incorrect, or empty. On average, each participant correctly answered 8.1 questions out of 10 (standard deviation 1.8), and incorrectly answered 1.4 questions (standard deviation 1.6). This result suggests that most of DNDRefactoring's mappings between drag-and-drop gesture to refactorings are relatively intuitive. Incorrect answers do not directly translate to unintuitive mapping. For example, for a question depicting "Extract Method" refactoring by dragging a set of statements from within a method and dropping it just above the method declaration (Figure 2a), at least two participants answered "creating a static block with selected statements". This particular mapping was overlooked during DNDRefactoring implementation because Eclipse does not support such refactoring, but we felt that the mapping is not counterintuitive.

We believe that empty answers are a stronger indication of the unintuitiveness of the drag-and-drop gestures to refactoring mappings. However, two questions with the most number of empty answers were the "Move" (Figure 2b) and "Extract Interface" (Figure 2c) refactorings. This was not surprising since many programmers do not use these refactorings frequently, and they are considered to be complex and moderately complex refactorings, respectively [26].

Figure 3: Survey scores out of 10, counting empty answers as incorrect answers.

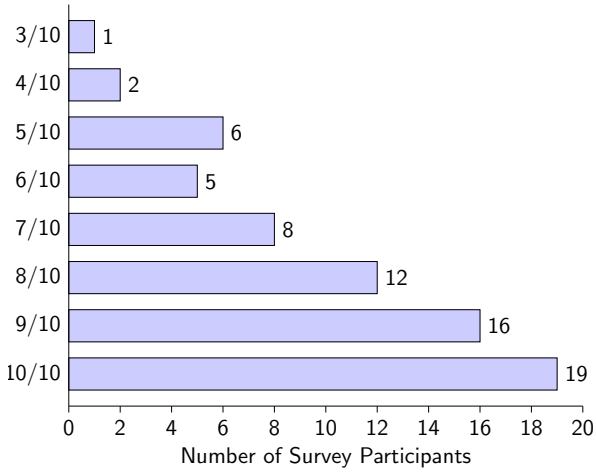


Table 3: Average number of answers per category per Eclipse competence level.

	Novice	Intermediate	Expert
Correct	7.7	8.1	9.2
Incorrect	1.4	1.5	0.8
Empty	0.93	0.44	0

We noticed a distinct trend in survey scores in relation to participants' competence in Eclipse refactoring engine. The number of correct answers increased as participants' competence in Eclipse refactoring engine increased, while the number of empty answers decreased (Table 3). Such result was expected, as expert users of Eclipse are probably more familiar with more refactorings supported in Eclipse than novice users.

Lastly, participants performed better in the second half of the survey (mapping from refactoring to drag-and-drop) than in the first half (mapping from drag-and-drop to refactoring). On average, more participants answered correctly in the second half (60.6) than in the first half of the survey (50.8), and there were less number of incorrect answers in the second half (5) compared to the first half (17). Both parts had the same average number of empty answers (3.4).

5.2 Lab Experiment Results

Table 4 shows the result of the controlled lab experiment. The *Invocation* column shows the three invocation methods tested, refactoring menus, keyboard shortcuts and DNDRefactoring (Menu, Shortcut, and DND, respectively). The first author used only the designated invocation method to perform all refactoring tasks except for the "Rename" refactoring which DNDRefactoring does not support, for which she used the menu invocation.

As Table 4 shows, DNDRefactoring performed significantly

Table 4: Results of the lab experiment. The experiment was repeated three times *per invocation method* for normalization (numbers in brackets show data obtained in each round, with minimum value italicized).

Refactoring	Invocation	KC[1]	MC[1]	MM (ft)[1]	Time (s)
1.Extract Method	Menu	(*19, 19, 19*) 19	(22, 22, *21*) 21.67	(12.57, *11.29*, 13.94) 12.60	(35.5, 34.5, *33*) 34.33
	Shortcut[2]	(*30, 30, 30*) 30	(14, *13, 13*) 13.33	(6.73, *6.50*, 6.66) 6.63	(*28*, 28.5, *28*) 28.17
	DND	(*11, 11, 11*) 11	(*12*, 13, 13) 12.67	(5.81, 6.23, *5.68*) 5.91	(*20.5*, 21.5, *20.5*) 20.83
2.Move Methods	Menu	(*0, 0, 0*) 0	(*24, 24, 24*) 24	(14.44, 16.27, *13.88*) 14.86	(30, *29.5*, 31.4) 30.3
	Shortcut	(*12, 12, 12*) 12	(16, *15, 15*) 15.33	(9.88, 9.55, *9.51*) 9.65	(26, *24.5, 24.5*) 25
	DND	(*0, 0, 0*) 0	(*10, 10, 10*) 10	(*6.33*, 6.46, 6.89) 6.56	(17.5, 17, *16.5*) 17
3.Anonymous to Top Class	Menu	(7, 7, *6*) 6.67	(*18, 18*, 19) 18.33	(*9.35*, 9.71, 10.40) 9.82	(*22.5*, 25, 24) 23.83
	Shortcut	(*15*, 16, 16) 15.67	(13, 13, *12*) 12.67	(7.48, 7.05, *6.82*) 7.12	(22.5, *20*, 21) 21.17
	DND	(*14, 14, 14*) 14	(*14, 14, 14*) 14	(*2.99*, 3.12, 3.08) 3.06	(*16*, 17, 17) 16.67
4.Extract Class	Menu	(*0, 0, 0*) 0	(*11, 11*, 12) 11.33	(*4.40*, 4.56, 4.76) 4.57	(14, *13, 13*) 13.33
	Shortcut	(*3, 3, 3*) 3	(*9, 9, 9*) 9	(*3.58*, 3.67, 3.74) 3.66	(*9*, 9.5, 9.5) 9.33
	DND	(*1, 1, 1*) 1	(*8, 8, 8*) 8	(1.94, 2.07, *1.90*) 1.97	(*7.5, 7.5, 7.5*) 7.5
TOTAL (AVERAGE)	Menu	25.67 (6.42)	75.33 (18.83)	41.84 (10.46)	101.79 (25.45)
	Shortcut	60.67 (15.17)	50.33 (12.58)	27.06 (6.77)	83.67 (20.92)
	DND	26 (6.5)	44.67 (11.17)	17.49 (4.37)	62 (15.5)

[1] KC = keyboard stroke counts, MC = mouse button counts, MM = distance covered by mouse movement.
[2] Keyboard shortcuts were 3-key combination, e.g. option-command-KEY in Mac, as is usually the default case.

better or at least comparably to other invocation methods in all cases. On average over 4 different refactoring tasks, DNDRefactoring required 0.08 more keyboard strokes (1.3% increase), 7.7 less mouse clicks (40.7% decrease), and 6.1 feet shorter mouse movement (58.2% decrease) compared to Menu invocation. Similarly, comparing DNDRefactoring to Shortcut invocation, DNDRefactoring on average required 8.67 less keyboard strokes (57.1% decrease), 1.4 less mouse clicks (11.2% decrease), and 2.4 feet shorter mouse movement (35.3% decrease).

5.3 Controlled User Study Results and Observations

Table 5 shows the each user study participant's results, collected over all refactoring tasks. There were two outlying cases where the participant #5 and #11 each introduced a fault when using Eclipse that caused a unit test to fail. They attempted to fix the fault manually and thus produced extremely high keyboard and mouse clicks and mouse movement values for Eclipse compared to DNDRefactoring. We felt that while these cases may give an insight to the complexity of current refactoring invocation tools in Eclipse, it is not a fair representation of them. Therefore the data from participant #5 and #11 were disregarded in the following analysis.

On average, each participant made 144.7 keyboard strokes when using Eclipse and 118.1 keyboard strokes when using DNDRefactoring. The average number of keyboard strokes saved by using DNDRefactoring was 26.6, (18.4%). Similarly, participants on average clicked the mouse button 101.6 times when using Eclipse and 63.1 times when using DNDRefactoring. The average number of mouse button clicks saved was 38.5 (37.9%). Lastly, participants' average mouse movement covered 62.7 feet when using Eclipse and 41.4 feet when using DNDRefactoring. The average distance saved was 21.3 feet (34.1%). Significant decrease in mouse button clicks and shorter mouse movement with DNDRefactoring, compared to Eclipse, was expected because DNDRefactoring eliminates the need for participants to open either the toolbar or mouse menus for refactoring, for which many programmers use their mouse. Many participants moved their mouse to find a specific refactoring in the menu as well. DNDRefactoring having no modal windows for configuration also resulted in less mouse usage. The more configuration items contained in the modal window means more navigation is required. Even small movements and occasional mouse button clicks can add up if there are a series of refactoring tasks. Relatively smaller decrease in keyboard stroke counts for DNDRefactoring compared to Eclipse was not surprising, as drag-and-drop is exclusively a mouse action. A possible reason for the reduction of keyboard strokes might be because programmers using DNDRefactoring tend to use their mouse when selecting Java elements to refactor, whereas many programmers used keyboards (shift + arrow keys) to make a selection.

An interesting trend we noted among almost all participants is that they made changes to their code in a uniform way. For example, if a participant manually changed the access modifier of a method when using Eclipse, she made the same decision when using DNDRefactoring, and vice versa. This suggests that DNDRefactoring does not cause drastic changes in participants' programming habits and thus the decreases in programming effort are unbiased.

We observed each user study participants during their sessions and recorded the number of times they encountered a *failure*. We define *failure* to include wrong refactoring selection, cancellation or undo of any refactoring performed, invoking a refactoring with an inapplicable Java element, or when refactoring invocation does not match programmers' expectations. The participants encountered the most number of failures when invoking "Move" refactoring with Eclipse. Eclipse's "Move" refactoring window shows a list of relevant objects, with their instance name and type, one from which a programmer can choose to move a method or field to. Many participants found the list confusing, and 6 of

Table 5: Results of the controlled user study.

		#1	#2	#3	#4	#5[1]	#6	#7	#8	#9	#10	#11[1]	Average[1]
KC[2]	Eclipse	136	296	84	92	469	445	23	180	82	109	405	144.7
	DND	153	244	151	72	74	320	30	50	40	121	155	118.1
MC[2]	Eclipse	118	112	120	95	424	55	103	109	88	216	237	101.6
	DND	75	79	122	74	112	33	59	45	50	94	131	63.1
MM[2](ft)	Eclipse	30.91	28.48	85.43	72.83	94.75	166.67	46.88	77.53	49.28	68.83	76.90	62.7
	DND	23.06	15.62	83.96	49.21	40.94	116.27	37.24	29.27	33.60	25.30	42.98	41.4
Failures	Eclipse	0	2	1	3	4	4	7	6	2	7	6	3.6
	DND	0	0	0	0	0	0	0	2	0	0	3	0.2

[1] Introduced a bug while refactoring with Eclipse, thus their data is not included in calculation of average.
[2] KC = keyboard stroke counts, MC = mouse button counts, MM = distance covered by mouse movement.

them canceled the refactoring up to 6 times. Selecting the right Java element to invoke specific refactorings was also failure-prone with Eclipse. For example, in order to extract a data class with a subset of fields declared in a class, 6 participants selected only the relevant fields in a Java editor and invoked the "Extract Class" refactoring, but Eclipse by default selects all available fields which negated the participants' preliminary actions. At least one participant did not notice that Eclipse had selected all fields and proceeded, eventually undoing the refactoring. With DNDRefactoring, two participant selected wrong drop targets while performing "Extract Method", "Convert Anonymous Class to Nested", and "Extract Data Class" refactorings.

The error case of participant #11 was what programmers would consider a usual programming mistake (incorrect string passed as an argument to a newly extracted method), but that of participant #5 provided an insightful opportunity to observe how programmers may introduce bugs while interacting with Eclipse's refactoring interfaces. We were able to retrace and replay her refactoring actions by using Eclipse's refactoring history [7] and interviewing her after the user study. The bug was introduced while she was moving a method from one class to another, and when one of the references to the moved method was not updated. She invoked the "Move" refactoring and followed the modal instructions, and opted to view the preview of the changes. Eclipse's refactoring preview window shows a list of Java source files that will be changed by the current refactoring, and allows programmers to exclude any file from the changes. During the interview, participant #5 stated that she remembers seeing one of the files being excluded (seemingly by default) but did not correct it. Upon replaying her refactoring history we concluded that the exclusion of a file was indeed the source of the bug, but also confirmed that Eclipse by default does *not* exclude any file from the change list. We concluded that she had mistakenly or unconsciously excluded a file but because it appeared to be a default setting, she accepted it to be correct. This case supports one of Murphy-Hill et al.'s findings, that majority of programmers do not modify default refactoring configurations, and we believe that this is one of the very issues that DNDRefactoring rectifies. DNDRefactoring uses the default refactoring configurations and thus streamlines the refactoring process, and do not burden the programmers or provide an opportunity for accidental bugs.

5.4 Post-Study Qualitative Survey

We asked each user study participant to answer a qualitative survey after they completed their tasks. Of the 11 user study participants, 9 found their interaction with DNDRefactoring to be very satisfactory, and 2 found it somewhat satisfactory. Also, 6 participants answered that DNDRefactoring was very comfortable to use while 5 reported that it was somewhat comfortable, and 8 participants found the translation from drag-and-drop to refactorings as expected but 3 found at least one of the refactorings unexpected (refactoring for extracting a data class), or the occasional lack of immediate in-line edit support a little cumbersome. We plan to mitigate these issues in the future, as detailed in the Section 8. All 11 would recommend DNDRefactoring to other people and some also suggested that it should be included as part of the Eclipse IDE. Some participants stated that DNDRefactoring "[is] very intuitive especially without knowing what the refactoring jargon means" and "saves me the trouble of remembering the exact refactoring to invoke", and that they "liked that several collated refactorings were invoked with a single action."

6. LIMITATIONS

DNDRefactoring at its current design state is not without its limitations. First and foremost is the difficulty of translating some refactorings into drag-and-drop gestures. Currently DNDRefactoring only supports move and extract-based refactorings. It is difficult, for example, to translate "Rename" refactorings in drag-and-drop gestures. Secondly, perhaps mirroring the first limitation, is that some drag-and-drop gestures can be translated into multiple refactorings. For example, drag-and-dropping an expression from within a method to its declaring class can easily translate into both "Extract Method" and "Extract Constant" refactorings. In an effort to follow our initial design goal of not interrupting programmers during the execution of refactorings, we default to the "Extract Method" refactoring. We plan to support multiple refactorings in the future by, for example, prompting programmers with a small selection of refactoring that they can choose from when they drop their drag source. Lastly, DNDRefactoring does not give programmers an option to see a preview of their changes. Recent studies have shown that many programmers in fact do not utilize the preview function in Eclipse [26]. There are a few implementation options to remedy this limitation, for example, giving users a switch to optionally turn on or off the preview func-

tionality or generating a set of all the changes performed that they can browse.

Some other limitations are specific to our current implementation of DNDRefactoring in Eclipse. Firstly, it currently lacks discoverability since there are no visual cues that suggest drag-and-drop gestures. We plan to include either a user manual or a short animation in Eclipse so that the new feature is easily discoverable. Secondly, it is not possible to drag-and-drop AST elements between the Java Editor and Outline View or Package Explorer. Supporting drag-and-drop between these different views can help make some of the drag-and-drop gestures easier to invoke. Currently, to perform an "Extract Method" refactoring, the user has to drag a statement from inside an existing method and drop it outside. This is problematic when the existing method is long since the user has to scroll to reach the boundaries of the method. By supporting drag-and-drop from the Java Editor to the Outline View, the user can conveniently drop the selected statements to a position in the Outline View without tedious scrolling in the editor.

While useful, drag-and-drop also has some shortcomings. One of the major concerns with drag-and-drop is that the entire gesture has to be completed in a single motion. This can be problematic when the drag source and drop target are obscured in the user interface, e.g. when the users operate on a smaller screens. Suspendable drag-and-drop techniques such as *Boomerang* alleviate this by allowing the user to first select the drag source, interact with other program elements and resume the drop gesture later [16]. Drag-and-drop can also be problematic on larger screens where the mouse has to travel further distances. *Pick-and-drop* alleaviates this by dynamically clustering and displaying the potential drop targets close to the mouse cursor *after* the source target has been selected [10]. Many other extensions are possible. Collomb and Hascoët provide a good introduction to other possible extensions and show how they can be unified to support different use cases [9]. Future work on DNDRefactoring could incorporate some of these extensions to make it easier to use on smaller or larger screens.

7. RELATED WORK

Drag-and-drop interfaces have traditionally been used in visual programming environments such as Alice [11], EToys [6] and Scratch [17]. In such environments, novice programmers write programs using visual blocks instead of text. Programmers use drag-and-drop as the primary means for organizing and restructuring those visual blocks.

Because visual blocks can be clunky to navigate in large programs, we eschew this approach in DNDRefactoring and implemented it directly in the textual Java editor, Package Explorer, and Outline View. Moreover, simple restructuring of visual blocks merely moves blocks to different locations in the program without considering behavior preservation. DNDRefactoring, on the other hand, intuitively maps each drag-and-drop operation to a corresponding refactoring operation that, when performed, preserves program behavior.

The typical modal window-based approach to invoking and configuring refactorings was introduced in the first refactoring tool, i.e. the Refactoring Browser [25]. For nearly two decades, little has changed in the interface of refactoring tools. Recently, Murphy-Hill et al. introduced new approaches to invocation with selection assists [18] and gesture-to-refacto-ring mappings [21]. Eclipse and IntelliJ have also introduced in-place refactoring features [1] that allow widely-used refactorings to be configured directly in the editor without the need for a modal window. Commercial tools such as CodeRush with Refactor! Pro [4] also aid programmers' refactoring tasks with suggestions and visual hints within the code, without modal windows. Nonetheless, these new approaches still rely exclusively on keyboard shortcuts and mouse menus. Our work investigates and demonstrates the potential of new methods of invocation for refactoring tools.

While drag-and-drag infrastructure has always been available in modern IDEs, none have truly exploited its capabilities. Existing IDEs such as Eclipse, NetBeans and IntelliJ provide *minimal* support for drag-and-drop refactoring. Currently, the only refactoring supported is "Move" refactoring, which can be invoked by drag-and-dropping a class into a package in the Outline View. All other drag-and-drop operations are interpreted as plain textual moves. Existing products dedicated to restructuring code only target organizational refactorings between different packages. For instance, Restructurer101 [5] provides a graphical view of all the classes and packages in the system and allows a developer to perform "Move" refactorings on them via drag-and-drop. To the best of our knowledge, our tool is the *first* to leverage the drag-and-drop as an intuitive way to invoke a variety of refactorings beyond move refactoring.

8. FUTURE WORK

Current implementation of DNDRefactoring assumes that programmers can accurately distinguish between different AST elements. We believe selection assist tools such as [18] will be an effective complement to DNDRefactoring. Also, visual cues such as highlights or tooltips indicating the specific refactoring that will be invoked may help narrow down programmers' selection of drop targets.

Apart from possible functional features for future versions of DNDRefactoring, we plan to apply the idea of programming by gestures and actions in different aspects of software engineering and evaluate its effectiveness. Since the action of drag-and-drop is more intimate and interactive, we conjecture that the use of DNDRefactoring during pair programming will be very helpful. During pair programming, an agile software development technique where two programmers work together at one workstation, the driver obviously has more control over the code changes then the observer. While this is expected, the driver's action of drag-and-drop may be easier and more intuitive for the observer to follow and understand. We also believe the drag-and-drop refactoring would be an effective tool in teaching refactoring. We are interested in impact of the difference in perception of the program - as a malleable entity instead of textual representation of a program - when novice programmers learn refactoring.

One of the critiques we received from user study participants and colleagues was the fact that some programmers are less inclined to use a mouse during programming. While we see this as no strict limitation of our tool, we recognize it as a possible barrier for some programmers to adopt DNDRefactoring. One possible remedy to this issue is to utilize a completely new technology, one of which is a touch screen. Eclipse, as of the Indigo version, does not support touch screen functionality, but we see a potential in implementing DNDRefactoring for the touch screen interface. We hypothesize that touch screens will provide even more inti-

mate and hands-on programming experience.

In addition, we plan to conduct a long-term study to analyze and evaluate the utility of DNDRefactoring in assisting programmers with floss refactorings. Would programmers using DNDRefactoring use the refactoring tool in IDE more often? If so, what kind of refactorings would they use DNDRefactoring for? We plan to collect refactoring data from programmers using DNDRefactoring in the wild, using such tools as [26], and draw correlations between floss refactoring and usage of DNDRefactoring.

9. CONCLUSIONS

We presented DNDRefactoring, a novel technique and tool that allows programmers to refactor their code by moving AST elements of the program, eliminating much of the interfaces that programmers have to understand and interact with. Our evaluation shows that DNDRefactoring is intuitive, and reduces programming effort of programmers. We believe that by using DNDRefactoring, programmers can more easily and intuitively refactor their code without resorting to manual code transformations, which in turn promotes the automated refactoring tools in IDEs.

10. ACKNOWLEDGMENTS

We thank Darko Marinov, John Brant, and Mohsen Vakilian for their valuable help and feedback, and Milos Gligoric and Jeff Overbey for participating in the initial user study. We also thank all the user study and survey participants. This work is dedicated to the loving memory of Brett Daniel.

11. REFERENCES

[1] In-place Refactorings. `http://www.jetbrains.com/idea/webhelp/editor.html`.

[2] Quick Assist. `http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.user/reference/ref-java-editor-quickassist.htm`.

[3] Refactor Actions. `http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm`.

[4] Refactor! Pro. `http://devexpress.com/Products/Visual_Studio_Add-in/Coding_Assistance/refactor_pro.xml`.

[5] Restructurer101. `http://www.headwaysoftware.com/products/`.

[6] Squeakland: Home of Squeak Etoys. `http://www.squeakland.org/about/`.

[7] Tips and Tricks (JDT). `http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.user/tips/jdt_tips.html`.

[8] WhatPulse. `http://whatpulse.org/`.

[9] M. Collomb and M. Hascoët. Extending drag-and-drop to new interactive environments: A multi-display, multi-instrument and multi-user approach. *Interact. Comput.*, 20(6):562–573, Dec. 2008.

[10] M. Collomb, M. Hascoët, P. Baudisch, and B. Lee. Improving drag-and-drop on wall-size displays. In *Proceedings of Graphics Interface 2005*, GI '05, pages 25–32, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[11] M. J. Conway. *Alice: Easy-to-Learn 3D Scripting for Novices*. PhD thesis, University of Virginia, 1997.

[12] S. Demeyer, M. Rieger, B. Van Rompaey, and B. Du Bois. Refactoring Lab Session. `http://lore.ua.ac.be/Research/Artefacts/refactoringLabSession/`.

[13] M. Fowler. Refactoring Home Page. `http://martinfowler.com/refactoring`.

[14] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[15] R. M. Fuhrer, M. Keller, and A. Kiezun. Advanced refactoring in the eclipse jdt: Past, present, and future. In *WRT*, pages 30–31, 2007.

[16] M. Kobayashi and T. Igarashi. Boomerang: suspendable drag-and-drop interactions based on a throw-and-catch metaphor. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, pages 187–190, New York, NY, USA, 2007. ACM.

[17] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010.

[18] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *In ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 421–430, 2008.

[19] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 2008.

[20] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[21] E. R. Murphy-Hill, M. Ayazifar, and A. P. Black. Restructuring software with gestures. In *VL/HCC*, pages 165–172, 2011.

[22] A. O'Connor, M. Shonle, and W. Griswold. Star diagram with automated refactorings for eclipse. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, eclipse '05, pages 16–20, New York, NY, USA, 2005. ACM.

[23] W. Opdyke and R. E. Johnson. Refactoring, an aid in designing application frameworks and evolving object-oriented systems. In *Proceeding of Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPA)*, 1990.

[24] C. Parnin, C. Görg, and O. Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM symposium on Software visualization*, SoftVis '08, pages 77–86, New York, NY, USA, 2008. ACM.

[25] D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 1997.

[26] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE, 2012. To appear.